Resource Availability-Aware Advance Reservation for Parallel Jobs with Deadlines

Bo Li[a,*], Yijian Pei[a], Bin Shen[b], Hao Wu[a], Min He[a], Jundong Yang[a]

[a]*School of Information Science and Engineering, Yunnan University, Kunming 650091, China*
[b]*School of Electrical and Information Engineering, Wuhan Institute of Technology, Wuhan 430073, China*

Abstract

Advance reservation is important to guarantee the quality of services of jobs by allowing exclusive access to resources over a defined time interval on resources. It is a challenge for the scheduler to organize available resources efficiently and to allocate them for parallel AR jobs with deadline constraint appropriately. This paper provides a slot-based data structure to organize available resources of multiprocessor systems in a way that enables efficient search and update operations, and formulates a suite of scheduling policies to allocate resources for dynamically arriving AR requests. The performance of the scheduling algorithms were investigated by simulations with different job sizes and durations, system loads and scheduling flexibilities. Simulation results show that job sizes and durations, system load and the flexibility of scheduling will impact the performance metrics of all the scheduling algorithms, and the $PE\ WorstFit$ algorithm becomes the best algorithm for the scheduler with the highest acceptance rate of AR requests, and the jobs with the $FirstFit$ algorithm experience the lowest average slowdown. The data structure and scheduling policies can be used to organize and allocate resources for parallel AR jobs with deadline constraint in large-scale computing systems.

Keywords:
multiprocessor, advance reservation, data structure, scheduling algorithm, deadline

---

[*]Corresponding author
  Email address: `libo@ynu.edu.cn` (Bo Li)

1. Introduction

Grid-like massive Internet computing platforms have emerged as an essential infrastructure for scientific and commercial applications and made it possible for "flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources"[1]. In order to guarantee the QoS of Grid applications in such environments, advance reservation(AR) is incorporated into many Grid systems, such as GARA[1], Nimrod/G[2] and G-QoSM[3], and many mainstream parallel scheduler, such as Maui[4], load sharing facility(LSF)[5] and protable batch system(PBS)[6]. Advance reservation makes the execution of grid jobs more predictable by reserving a particular resource capability over a defined time interval on local resources, and has been widely used for many kinds of jobs, such as single processor jobs[7], parallel jobs[8], mixed-parallel applications[9], co-allocation jobs[10], bag-of-tasks[11], and workflow[12]. Advance reservation has been an important area of interest in the Grid community.

In order to reserve resources in such AR systems, a user must submit a request to the system by specifying a set of parameters such as number of processing elements needed,ready time, duration and deadline. For such an AR request, the job cannot start until its ready time and it must be completed by its deadline. Upon receiving such an AR request, it's the task of the scheduler to decide if there are sufficient available resources such that the request can be completely executed within the interval of its ready time and deadline. Considering that there are so many resources and optional allocations to check in a large-scale computing system, it's quite a challenge for the scheduler to organize available resources efficiently and to allocate resources for dynamically arriving AR requests appropriately: the scheduling procedure itself will impact the ability and efficiency of the scheduler to manage resources and to schedule a great number of jobs with various requirements, and the scheduling decision will also impact the performance perceived by users and service providers. For users, the fraction of AR requests accepted by the scheduler and the turnaround time are important measures of how well their service requests are treated[13, 14]. For clusters, the acceptance of new reservations will fragment a continuous range of resource into pieces, and thus reduce the potential scheduling opportunities and results in lower utilization[15, 16]. The key challenges here lies in two aspects: (1) to de-

velop efficient data structure to organize available resources for AR requests in a way that enables efficient search and update operations; (2) to develop a group of scheduling algorithms or policies that improve the performance perceived by users and providers.

In the literature, many data structures(such as array[17], linked-list[18], trees[19, 17, 20] and queues[21, 22]) and scheduling algorithms([23, 15, 24, 20]) have been proposed for advance reservations and widely studied. However, these data structures and scheduling algorithms are only suitable for single- or multi-processor AR jobs with immediate deadline constraint. For such kind of AR jobs, they must be scheduled to run at their ready time and their deadlines are immediate(i.e., deadline=ready time+duration). However, if the AR requests are not strictly asked to begin to run at their ready times, they can begin to run at any time within its ready time to its latest start time(i.e., =deadline-duration). Such kind of AR request is more general than those with immediate deadlines, and makes it more flexible and complicated for the scheduler to organize available resources, to control admission and to schedule. As a result, all those existing data structures and scheduling algorithms for AR requests with immediate deadlines are not suitable for these with general deadlines.

Up to now, only few researches have been done for AR jobs with general deadlines. In [25] and [7, 26], the authors investigated the problem of how to allocate a single-channel(or single-processor) AR job with general deadline constraint to n single-processor servers. In those works, because each job only needs to be reserved on a single-processor server, it is not necessary to allocate more than one idle intervals across multiple servers for them simultaneously, thus all the algorithms in those works only considered the temporal constraint, without considering the scheduling of AR jobs with more than one resources simultaneously.

Despite the fact that existing data structures and algorithms have been widely used for AR requests, they are not suitable for parallel AR jobs with general deadlines in large-scale multiprocessor systems. In this paper, we investigated the problem of how to manage and allocate multiprocessor resources for parallel AR jobs with general deadline constraint. Different from existing data structures and scheduling policies designed for scheduling single-processor deadline-constraint AR jobs to n single processor servers, or for scheduling parallel AR jobs with immediate deadlines to a multiprocessor system, in this work we proposed a new data structure and scheduling policies to organize the availability of resources in a large-scale multiprocessor

3

system and to allocate them appropriately for parallel AR jobs with general deadline constraint. The main contribution of this work include:

- Proposed a new data structure to organize available resources efficiently in multiprocessor systems for *single- or multiple-processor* AR requests with *immediate or general* deadline constrains;

- Proposed a set of operations for the data structure to enable efficient search and update operations;

- Proposed a set of scheduling policies for the data structure to allocate resources for AR requests, and investigated their performance via simulation. New scheduling policies can be added into the data structure flexibly.

The rest of this paper is organized as follows. We discuss related work in Section 2 and describe the model for scheduling parallel AR jobs with general deadlines in a multiprocessor system in Section 3. In Section 4 we introduce a slot-based data structure to organize the availability of resources in a way that enables efficient adding, deleting and searching operations. In Section 5 we provide a suite of scheduling algorithms for parallel AR requests with general deadline constraint and present a comprehensive performance evaluation study of the algorithms by simulations, and we conclude the paper in Section 6.

2. Related work

Many data structures and scheduling algorithms have been proposed for advance reservations. Most of them are suitable for AR requests with immediate deadlines, and only few of them were specifically designed for AR requests with general deadlines. For AR request with immediate deadlines, such data structure as array[17], linked-list[18], trees[19, 17, 20] and queues[21, 22] have already been widely studied. These data structures are primarily used for admission control and focused on finding out whether it's feasible for the scheduler to accept an AR request to start at a definite time and keep on running for a given period. In [22] the author presents a good summary and comparison of them when they are used for single- or multi-processor AR jobs with immediate deadline constraint. However, they are not specifically designed for AR requests with general deadline constraint.

Based on existing scheduling theory and algorithms for jobs with or without deadlines, some variants of scheduling algorithms for jobs with advance reservations have been studied in Grid-like systems[23, 15, 24, 20] and their impact on the users and the systems were investigated in terms of turnaround time, slowdown, or utilization.

Different from existing plentiful researches for AR requests with immediate deadlines, only few works have been done for AR jobs with general deadlines. In [25],the problem of how to reserve optical bursts on wavelength channels whose bandwidth may become fragmented with idle intervals was proposed. By using concepts from computational geometry,the author maps each idle interval and each burst as a point on a two-dimensional plane,then the points for idle intervals were organized into a search tree and several algorithms,such as Min-SV, Max-SV, Min-Ev, Max-EV and Best-fit,were proposed for reserving bursts with and without fiber delay lines. Based on the concept and algorithms in [25], in [7, 26] the author adapted them for scheduling *single-processor* AR jobs with general deadline constraint to n *single-processor servers*. In those works, because each job only needs to be reserved on a single-processor server, it is not necessary to allocate more than one idle intervals across multiple servers for them simultaneously, thus all the algorithms in those works only considered the temporal constraint, without considering the scheduling of AR jobs with more than one resources simultaneously. Moreover, for different scheduling policies in [7, 26], the data structure used for storing the availability information of the resources and the method for finding out the appropriate interval are different. This limits the flexibility of the data structures to support new scheduling policies. In contrast, the data structure proposed in this paper can support different scheduling policies flexibly, without changing the data structure itself and the method of finding appropriate resources for the requests.

3. Problem description

The computing environment is a parallel system, e.g., clusters or massively parallel processing machines, consisting of a group of space-shared processing elements $\{PE_1, PE_2, ..., PE_n\}$, with the total number of n. For simplicity, we assume the PEs are homogeneous. Each machine has a local resource management system capable of supporting advance reservations for local or external jobs. Figure 1 shows an example schedule of a parallel AR job with general deadline constraint. Assume the request of the AR job ar-

rives at t_0. The request asks the scheduler to allocate PEs during the ready time and the deadline so as the job can run for its duration. On receiving this request, the scheduler will evaluate whether they are enough resources available for the job so as to meet its deadline. If so, the scheduler will allocate and reserve them for the job; otherwise,the request will be declined. Moreover, if there are more than one allocations that can satisfy the request at the same time, only one of them will be chosen based on some criteria or policies.

In this paper, each AR request with deadline is characterized by a five-parameter tuple $(t_a, t_r, t_{du}, t_{dl}, n_{pe})$, where:

1. t_a is the arrival time of the request;
2. $t_r (\geq t_a)$ is the ready time, i.e., the earliest start time of the job. When $t_r > t_a$ is permitted, advance reservations are supported by the scheduler; Otherwise, only immediate reservations are permitted;
3. t_{du} is the duration of the job, i.e., the amount of time needed by the job when running on current cluster;
4. $t_{dl} (\geq t_r + t_{du})$ is the deadline, i.e., the latest time by which current job must be completed. If $t_{dl} = t_r + t_{du}$, the deadline is immediate and we refer to this problem as scheduling with immediate deadline. If $t_{dl} \geq t_r + t_{du}$, the deadline is general and we refer to this problem as scheduling with general deadline; and
5. n_{pe} is the number of PEs required by the request.

In Figure 1, at t_0, there are respectively two running jobs(job1 and job2) and one reserved job(job3) on the cluster. The scheduler can try to allocate the job to start at any time from the ready time (i.e., $t2$) to the latest start time, i.e., $t7(= t_{dl} - t_{du})$, and then check whether there are enough PEs for the job to begin to run at the selected start time for t_{du}.

In this paper, we assume all AR jobs arrive dynamically and they are non-preemptive and non-malleable, i.e., they must run till completion once they start execution and their requirements on resources, such as the number of PEs, can not be changed. Compared with preemptive and malleable AR jobs[15], such kind of non-preemptive and non-malleable AR jobs are more difficult to tackle for the scheduler. It appears to be NP-complete to schedule them under deadline constraint even for very restricted cases, and there are not optimal online scheduling algorithms for them[27]. In order to schedule this kind of AR jobs, heuristics are left for the scheduler.
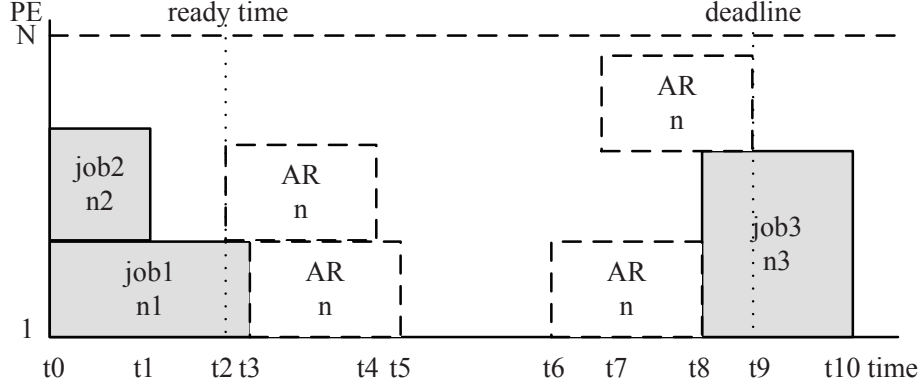
Figure 1: An example of reserving processing elements for a new advance reservation request with general deadline constraint. Assume the new AR request arrives at $t0$, when there are 2 running jobs(job1 and job2) and one reserved job(job3), and the AR request needs n processing elements and should be processed within its ready time($t2$) and its deadline($t9$). Four feasible allocation for the AR request are illustrated.

In the scheduling, both of the procedure to decide and the decision itself made by the scheduler are important. Because there are so many PEs and optional start times to check, the scheduling procedure itself will impact the ability and efficiency of the scheduler to manage a large-scale distributed resources and to schedule a great number of jobs with various requirements, and the scheduling decision will also impact the performance perceived by users and service providers. In the following sections, an efficient data structure and operations, and a suite of scheduling strategies will be proposed to manage and allocate resources for AR requests with deadline.

4. Data Structure and operations

Similar to the variable slot data structure in [28], for each cluster, we represent the resources allocated for each running or reserved job as an rectangle and record the availability of a cluster as a set of $\{time, PEs\}$ pairs, where $time$ means at which the state(i.e., busy or idle) of the PEs change, and PEs means the identities of the PEs who are busy at $time$. If PEs is null($\emptyset$), it means at that time all busy PEs recorded in the previous time slot are set free.

In order to record and manage the identity of PEs occupied in every time period, a linked list-based data structure *AvailRectList* was proposed.

When a new job is allocated with it start time(t_s), it end time(t_e) and its PEs(PE_{job}), the records within the interval $[t_s, t_e)$ will be updated by adding PE_{job} to their PEs, inferring that PE_{job} will be used by the job from t_s to t_e. Accordingly, when the job is completed, PE_{job} will be released and subtracted from the records within t_s to t_e. In this way, at any time, the PEs already occupied for running or reserved jobs are known, and we can check for the availability of PEs in any given time interval.

Additionally, to simplify the process of querying the time slots at which the states of PEs change and the availability of PEs, an auxiliary sorted set-based structure, *TimeSet*, was used. As the records in *AvailRectList* change, *TimeSet* will be updated synchronously.

In order to support advance reservations with deadline, the data structure needs to perform three basic operations: adding an allocation, deleting an allocation and searching for available allocations for AR requests.

4.1. Adding and deleting an allocation

Before adding an allocation, we assume a search operation(see Subsection 4.2) has already been done and the start time(t_s), the end time(t_e) and the PEs of the job(PE_{job}) have already been allocated. The adding operation is described in Algorithm *addAllocation*(t_s, t_e, PE_{job}). The core of this operation is to update the records in the interval $[t_s, t_e)$ by adding PE_{job} to their PEs. If *AvailRectList* is empty or the earliest time of the records is greater than t_e, just need to add $\{t_s, PE_{job}\}$ and $\{t_e, \varnothing\}$ into *AvailRectList*;Otherwise,we should find all records in the interval $[t_s, t_e)$ and update their PEs by adding them with PE_{job}(line 4-5). After updating, it is possible that the PEs of the record of t_s or t_e become the same as that of the record of the time slot just before, or that t_s is the earliest time slot and the PEs of the record of t_s are null. In such cases, the records of t_s and/or t_e are redundant and should be cleaned((line 7). When a job finishes, a deleting operation *deleteAllocation*(t_s, t_e, PE_{job}) is called immediately. It applies to the same principle as adding a new one but to update the records in the interval $[t_s, t_e)$ by subtracting PE_{job} from their PEs.

The complexity of the *addAllocation()* or the *deleteAllocation()* operation is analyzed as follows. Suppose there are n records in *AvailRectList*. For the *addAllocation()* or the *deleteAllocation()* operation, we need to update the records within $[t_s, t_e)$. Assume there are n' records within $[t_s, t_e)$ and k PEs will be updated in each record. It will take $O(n' * logn)$ time to find n' time slots by searching *TimeSet*, take $O(n)$ time to find the record for each

---
Algorithm 1: addAllocation(t_s,t_e,PE_{job})
---
1 if *AvailRectList is empty OR TimeSet.first > t_e* **then**
2 $\quad$ *AvailRectList.addall({t_s,PE_{job}},{t_e,$\emptyset$})*;
3 else
4 $\quad$ find all records within $[t_s, t_e)$ in *AvailRectList*;
5 $\quad$ update the PEs of the records found by adding them with PE_{job};
6 end
7 clean possible redundant records;

Algorithm 2: deleteAllocation(t_s,t_e,PE_{job})
---
1 find all records within $[t_s, t_e)$ in *AvailRectList*;
2 update the PEs of the records found by subtracting them with PE_{job};
3 clean possible redundant records;
---

time slot in the linked list and $O(k)$ time to update k PEs for each record. After updating the records of t_s and t_e, it will take $O(1)$ time to remove them if they are redundant. Thus the overall complexity of finding and updating n' records will take $O(n' * (n + k + logn))$ time.

4.2. Search feasible allocation

When a new AR request arrives, this operation is performed to check whether there are enough PEs to be allocated for the job. If so, the operation will choose and return the identities of allocated PEs and the start time for the job; Otherwise, it will return null, inferring that there are not enough PEs for the request. This operation is defined as $findAllocation(t_r, t_{du}, t_{dl}, n_{pe}, policy)$ and is shown in Algorithm 3, where *policy* is the scheduling policy used to choose available PEs and runtime intervals for the request(see Section 5). If *AvailRectList* is empty, the operation will allocate the request to start at t_r and allocate n PEs for it(line 2-3); Otherwise, the operation will search for feasible start times(line 5), get the maximum availability rectangle of every start time(line 6-9) and add them into *availRect*; If finally *availRect* is not empty, inferring there are feasible allocations for the request, the operation will choose an appropriate start time and allocate PEs for the request according to the scheduling policy(line 11).

Notably, any time slot in the interval $[t_r, t_{dl}]$ may be an optional start time for the request. This makes it a hard work to check the availability

9

rectangles of resource related to these start times. To simplify this operation and to minimize the possible fragmentation of resources resulting from AR allocations, in the operation of line 5, it's suggested to check existing time slots only in the interval $[t_r, t_{dl}]$ and new ones generated by deducting these existing time slots with t_{du}. For every optional start time t_s, the operation gets free PEs(i.e., PE_{free}), in the interval $[t_s, t_s + t_{du})$. This can be done by iterating through *AvailRectList*. If the number of free PEs is not less than nPE, indicating t_s is a feasible start time, and the operation will find the maximum availability rectangle containing PE_{free} and the interval $[t_s, t_s + t_{du})$(line 7). Finally, after constructing availability rectangles for all feasible start times, the operation will choose one of them according to *policy*, and return the appropriate start time and n_{job} PEs for the request.

Algorithm 3: *findAllocation(t_r,t_{du},t_{dl},n_{job},policy)*

---

1 **if** *AvailRectList is empty* **then**
2   Let the job to run at t_r and allocate n PEs for it;;
3   return $\{t_r, \text{IDs of the } n \text{ PEs}\}$;
4 **else**
5   find all feasible start times $\{ST\}$ within $[t_r, t_{dl} - t_{du}]$;
6   **foreach** *element t_s in ST* **do**
7   find the maximum availability resource rectangle $\{T_{begin}, T_{end}, PE_{free}\}$ of t_s;
8 $availRect.add(\{t_s, T_{begin}, T_{end}, PE_{free}\})$;
9   **end**
10   **if** *availRect is not empty* **then**
11   choose the appropriate t_s and n_{job} PEs according to *policy* from availability resource rectangles, and return $\{t_s, \text{IDs of the } n_{job} \text{ PEs}\}$;
12   **else** return null;
13 **end**

The complexity of *findAllocation()* is as follows. If *AvailRectList* is empty, the request will be allocated to run at t_r immediately. This will take $O(1)$ time; Otherwise, we can sort the linked list into sorted array list(this will take $O(nlogn)$), and assume there are p feasible start times within $[t_r, t_{dl}]$. For each feasible start time, assume there are u free PEs in its maximum availability rectangle and v neighboring records should be checked to deter-

mine the maximum availability rectangle, it will take $O(u*v)$. After getting the maximum availability rectangle of each feasible start time, the information of the rectangles will be used to build a priority queue($O(p)$), in which the selected rectangle according to *policy* will always be in the root($O(1)$). Finally, a group of n_{job} free PEs will be chosen from the selected rectangle with u free PEs and allocated to the request($O(n_{job}*logu)$). Overall, the complexity of searching and allocating resources for the request will take $O(p*u*v + nlogn + n_{job}*logu + p)$.

The following example illustrates a typical application of these operations in Figure 1. At t_0, there are two running jobs and one reserved job. Assume the running jobs begin to run at t_0 and the records in *AvailRectList* are $\{t_0, n1 + n2\ PEs\}, \{t_1, n1\ PEs\}, \{t_3, \emptyset\}, \{t_8, n3\ PEs\}$ and $\{t_{10}, \emptyset\}$. The following steps illustrate the actions of the above operations.

(1)When a new AR request $\{t_2, t_4 - t_2, t_9, n\}$ arrives, the scheduler calls *findAllocation($t_2, t_4 - t_2, t_9, n, policy$)* to find available start times and free PEs for the job. Theoretically, it's optional for the AR job to start at any time slots within from the ready time t_2 to the latest start time t_7. However, we only choose t_2, t_3, t_6 and t_7 as feasible start times and neglect any other slots. In this way, we can simplify the searching operation and lower the influence of AR requests on fragmenting resources.

(2)Fortunately, there are enough free PEs for the AR request to begin to run at any of the four start times, and *findAllocation()* will calculate the maximum availability rectangle for every start time and choose one of them for the AR request. For t_2,the number of free PEs within the interval $[t_2, t_4)$ are $N - n1$, and the beginning slot and ending slot of the maximum availability rectangle with $N - n1$ free PEs are t_1 and t_8. For t_3, the number of free PEs within the interval $[t_3, t_5)$ are N, and the beginning slot and ending slot of the maximum availability rectangle with N free PEs are t_3 and t_8. In this way,we can get the numbers of free PEs and the beginning slots and ending slots of the maximum availability rectangles of t_6 and t_7 respectively. Assume *policy* is *PE Worst Fit*(see *Section*5) and t_3 is chosen as the start time and n PEs will be allocated for the request, the operation will return $\{t_3, n\ PEs\}$.

(3)After getting the start time and PEs for the AR request, the scheduler will call *addAllocation($t_3, t_5, n\ PEs$)* to add the reservation into *AvailRectList*. At first,the adding operation updates $\{t_3, \emptyset\}$ to $\{t_3, n\ PEs\}$, and inserts $\{t_5, \emptyset\}$ into *AvailRectList*. Because$\{t_1, n1\ PEs\}$ is the exactly previous record of $\{t_3, n\ PEs\}$ in *AvailRectList* and the $n1$ PEs of t_1 are the

same as the n PEs of t_3, $\{t_3, n\ PEs\}$ will be merged with $\{t_1, n1\ PEs\}$ and removed from *AvailRectList* .

(4)At t_1, job2 finishes, and *deleteAllocation*$(t_0, t_1, n2PEs)$ will be called to subtract $n2$ PEs from the records within the interval $[t_0, t1)$. The original record of t_0 will change from $\{t_0, n1+n2\ PEs\}$ to $\{t_0, n1PEs\}$, and the original record of t_1, i.e., $\{t_1, n1\ PEs\}$, will be merged with the new $\{t_0, n1PEs\}$ and then be removed. Finally the remaining records in *AvailRectList* are $\{t_1, n1PEs\}$, $\{t_5, \emptyset\}$, $\{t_8, n3\ PEs\}$ and $\{t_{10}, \emptyset\}$.

5. Scheduling algorithms

If there are more than one allocations that can satisfy the request at the same time, the scheduler will choose one of them based on some criteria. Considering feasible start times themselves and their maximum availability rectangles, we have developed following scheduling strategies to control the allocation of resources for AR requests.

First Fit(FF): the job is allocated to run at the earliest feasible start time.

PE Best Fit(PE_B): the job is allocated to run at the feasible start time with the minimum number of free PEs.

Duration Best Fit(Du_B): the job is allocated to run at the feasible start time, the availability rectangle of which has the minimum duration.

PE-Duration Best Fit(PEDu_B): the job is allocated to run at the start time, the availability rectangle of which has the minimum production of the number of free PEs and the duration.

Different from **PE_B**, **DU_B** or **PEDU_B** that tries to choose feasible start time the availability rectangle of which has the minimum number of free PEs, duration or production, we can also construct their corresponding maximum versions, i.e., the **PE Worst Fit (PE_W)** algorithm, the **Duration Worst Fit (Du_W)** algorithm and the **PE-Duration Worst Fit (PEDu_W)** algorithm.

In practice, it's possible that more than one feasible start times have the same availability rectangle. For example, in Figure 1, t_3 and t_6 have the same availability rectangle, which has N free PEs within t_3 and t_8. In such cases, if the maximum availability rectangle was chosen for the request, the earliest

feasible start time will be chosen, so as to shorten the waiting time of the job. e.g., in Figure 1, t_3, instead of t_6, will be chosen by the scheduler when Du_B or PE_W is used.

6. Performance evaluation

In order to verify the data structure and its operations, and to evaluate the performance of the scheduling strategies, we implemented the data structure and its operations in a discrete event-driven simulator, applied these strategies to schedule AR requests, and analyzed their performance metrics.

The simulator is implemented on the basis of SimJava[32], which is a process based discrete-event simulation package for Java and is originally developed by University of Edinburgh. For its accuracy in simulation, SimJava is widely used to build simulators in many researches. A SimJava simulation is a collection of entities each running in its own thread and they are connected together by ports and can communicate with each other by sending and receiving events. A central system class controls all the entities, advances the simulation time, and delivers the events. In our simulator, we implemented a hierarchal architecture to model cluster or grid-like computing environments and to evaluate the operation and performance of different resource management strategies. The simulator includes entities such as meta-users, meta-schedulers and multiprocessor systems. A meta-user is responsible for generating AR requests and submit them to the job queue of the meta-scheduler, and the meta-scheduler links to multiprocessor entities and manages their availability information via the data structure proposed in this paper and allocate resources according to the scheduling policies. In a multiprocessor entity, a local scheduler entity and multiple processing element entities were created and they are responsible for processing the AR request submitted by the meta-scheduler.

6.1. Simulation environments

For experiments based on discrete event-based simulation, a workload is needed to drive the simulation. However, there are not any workload traces about advance reservation can be used in this paper directly. In this paper, the LANL-CM5 in Parallel Workload Archive[30] and the Feitelson-Lublin model[31] were considered to generate AR jobs with deadline constraints. The LANL-CM5 is a 1024-node Connection Machine CM-5 system and processors are allocated only in powers of 2, with the minimal partition size

13

and the maximal partition size being 32 and 1024 processors respectively. In experiments, the distributions and parameters used in the Feitelson-Lublin model to generate workload were set according to the LANL-CM5 values in [31],following models and parameters were used to control the generating of jobs:

(1)The combined model of arrival process in the Feitelson-Lublin model and its parameters for LANL-CM5 were used to control the arrival of jobs.

(2)The two-stage uniform distribution with parameters ULow,UMed,UHi and Uprob was used to control the sizes of jobs generated. In this distribution,all jobs are parallel,i.e.,the probability for serial jobs are 0, and their sizes are power of 2,with the minimal size of 32(i.e.,ULow=4.5), the maximal size of 1024(i.e.,UHi=10) and $Uprob = 0.82$. For the original LANL-CM5 log,UMed is 7. In order to control the sizes of the jobs generated, UMed was set to be 5, 6, 7, 8, and 9 individually in experiments. As UMed changes from 5 to 9, the mean size of jobs increases.

(3)Runtime is an important characteristic of a rigid job. In the Feitelson-Lublin model, the hyper-Gamma distribution was used to model runtimes and a group of parameters were verified to be appealing and representative for each and all workloads. Although the resulting runtimes in this model are discrete, the distribution of which is very different from the distribution of sizes and spans a very large range of values. In the interest of efficient computability and representability, we made minor modifications for this model to only generate runtime values of 60, 300, 900, 1800, 3600 and 10800. The distribution of these new runtime values were determined by comparing the distribution of estimated runtimes in the original LANL-CM5 records and the distribution of runtimes generated in the model. Moreover, as the size and the runtime of a job are correlated, when UMed changes,the distributions of sizes and runtimes will change. In this way, we can evaluate the performance of different scheduling algorithms as the distributions of sizes and runtimes of jobs change.

By using the the LANL-CM5 workload and the Feitelson-Lublin model, we can generate a series of jobs, each with arrival time, size and duration. In order to add deadline and advance reservation constraints to the resulting jobs, two factors were used:

- artime factor(≥ 0): is used to control the period between the arrival time t_a and the ready time t_r of an AR request. The period is defined as $artime factor * U[0,1] * t_{du}$, where $U[0,1]$ is a random number uniformly

14

distributed in $[0, 1]$.. This parameter is set based on [33].

- deadline factor(≥ 0): is used to control the job's deadline, which is defined as $t_r + (1 + deadline\,factor * U[0, 1]) * t_{du}$. If this factor is zero, the deadline is immediate, i.e., $t_{dl} = t_r + t_{du}$; Otherwise, the deadline is general, i.e., $t_{dl} \geq t_r + t_{du}$.

With these parameters, we can generate jobs with deadlines and advance reservation requests from a workload trace. For AR jobs, as the values of *artime factor* and *deadline factor* increase, the flexibility of scheduling will increase, and the resource competition between AR jobs will be alleviated. Based on the influence of these parameters on AR jobs, these factors were combined together as $\{artime\,factor, deadline\,factor\}$, and five pairs of values, i.e., $\{1, 1\}, \{2, 2\}, \{3, 3\}, \{4, 4\}$ and $\{5, 5\}$, were used to generate low-, middle- and high-flexibility AR jobs.

In order to generate workloads with different distribution of inter-arrival times and further to investigate the performance of strategies under different system load, *arrival factor(af in short)* is defined and used as follows: for a job in a given workload with arrival time t_{s0}, its new arrival time will be $t_{s0}/arrival\,factor$. In this way, we can control the arrival of jobs and thus control system load. In experiments, $af = 1$ is set as default.

In experiments, following two metrics were used to evaluate the performance of different scheduling strategies:

(1)*Acceptance rate*: is the percentage of reservations that are accepted because their requirements can be satisfied, which indicates the ability to accommodate AR request.

(2)*Average slowdown*: the slowdown of an AR job is the response time of the job normalized by the running time, i.e., $(waiting\,time + runtime)/runtime$, where waiting time is the difference between the ready time and the actual start time. This measures how much slower the job ran due to conflicts with other competing jobs and it seems more reasonable than the waiting time to capture the user's expectation that a job's waiting time will be proportional to its runtime. Average slowdown is the average value of slowdowns of all accepted AR jobs, which indicate how well the scheduling algorithm can satisfy the user's expectations on the execution of the job.

6.2. Experimental results

In experiments, we investigated the performance of the scheduling strategies against different job sizes and durations, different arrival factors and dif-

ferent {*artime factor,deadline factor*} values.For each experiment, 10^4 jobs were submitted to the scheduler for the results, and we have obtained 95% confidence intervals for them.

6.2.1. Results for different job sizes and durations

Figure 2 and Figure 3 present the acceptance rate and the average slowdown of different scheduling strategies for workloads with different UMed values respectively. In experiments, *arrival factor* is 1 and {*artime factor,deadline factor*} is {3, 3} as default. As shown in Figure 2, when UMed changes from low to middle and high, the acceptance rates of all strategies decrease gradually. This result is in agreement with intuition: as UMed increases, the mean size and the mean duration of jobs increase, thus demanding more resources to accommodate them and intensifying competition of available resources between jobs.

In Figure 2, there are three groups of algorithm with almost identical behavior: PE_W and Du_B, $PEDu\_B$ and $PEDu\_W$, and PE_B and Du_W. Among them, except the PE_B algorithm and the Du_W algorithm, all other four strategies outperform *FF*. Moreover, the PE_W algorithm and the Du_B algorithm in the first group perform much better than *FF* and clearly become the best strategies for all UMed values. Notably, except for the almost identical behavior of $PEDu\_B$ and $PEDu\_W$, the performance of PE_B and PE_W, and Du_B and Du_W are quite different. Based on this results, we cannot draw a deterministic conclusion that PE-based strategies are better or worse than duration-based ones, or best fit-based strategies are always better or worse than equivalent worst fit-based ones. This can be explained by the fact that, for an idle period of resource,the influences of the number of its PEs and its duration on accommodating new jobs are different.

Now turn to Figure 3, which plots the average slowdown of different strategies for workload with different UMed values. When UMed changes from low to middle and high, the average slowdown of all strategies increase in general. This can be explained by the fact: as UMed increases, the mean size and the mean duration of jobs increase. For a new job,no matter under which algorithm, it will experience a longer waiting time before execution. As we can see, the jobs with FF experience the lowest average slowdown. This can be easily explained that FF always chooses the earliest feasible period and thus minimizing the waiting time of jobs. For the other strategies, the performance of PE_W and Du_B, and PE_B and Du_W are similar again as in Figure 2. However, the performance of $PEDu\_B$ and $PEDu\_W$ are
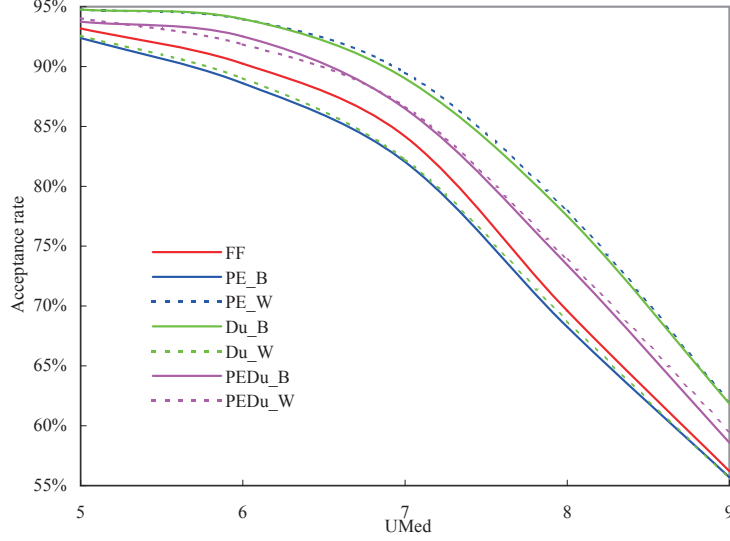
surprisingly different.

Figure 2: Acceptance rate vs job size control parameter UMed

6.2.2. Results for different system load

In both [31] and the aforementioned experiments,UMed is typically set to 7. In following experiments, we will investigate the performance of the strategies against different arrival factor values with UMed=7 and $\{artime\ factor, deadline\ factor\}=\{3, 3\}$.

Figure 4 and 5 illustrate the acceptance rate and the average slowdown of the strategies as arrival factor changes from 0.5 to 1.5, in step of 0.25.As expected, as the value of arrival factor increases, acceptance rates and slow-downs of all strategies degrade in both Figures.This agrees with the fact that as the value of *arrival factor* increases, the number of AR requests submitted within a given period will increase,thus the competition of resources among jobs will intensify,and the acceptance rate will decrease. For accepted AR requests, they also tend to experience longer waiting time,for there will be more jobs allocated in their expected execution periods as the value of *arrival factor* increases.

By comparing the results in Figure 2 and 4 and the results in Figure 3 and 5 respectively,it can be seen clearly that the relative performance of the scheduling algorithms in both experiments are similar. Based on the results
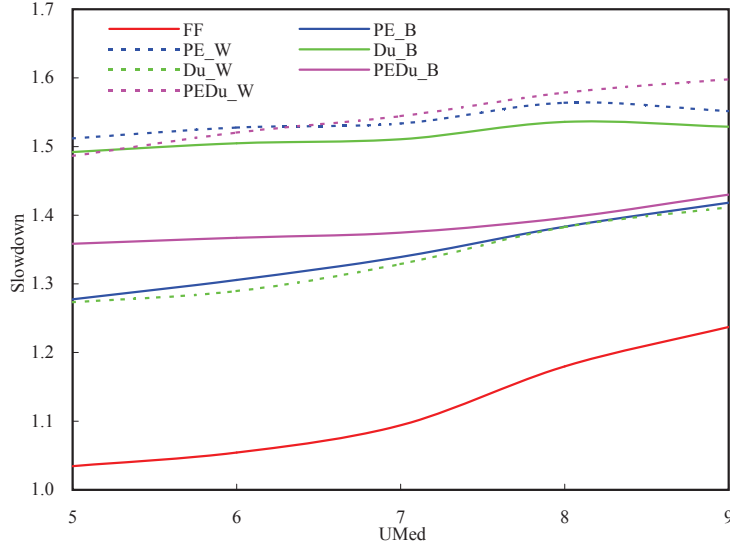
17

Figure 3: Slowdown vs job size control parameter UMed

in Figure 2-5, we can conclude that job sizes and durations and system load will impact the performance metrics of all the scheduling algorithms and the performance perceived by the users clearly: as job sizes and durations and system load increase, the acceptance rate and the average slowdown for all algorithms will degrade, and AR jobs will experience lower acceptance rate and higher waiting time.

6.2.3. Results for different scheduling flexibilities

Figure 6 and 7 present the acceptance rate and average slowdown of different scheduling algorithm when the values of {*artime factor,deadline factor*} change. As shown in Figure 6, when the values of {*artime factor,deadline factor*} change from low to middle and high, the acceptance rates of PE_W, Du_B and $PEDu\_B$ increase almost linearly. This behavior indicates that their acceptance ability are stable throughout the range of flexibilities considered in this study.However, the performance improvements of other four strategies are not stable, especially at $\{4, 4\}$, indicating that they are sensitive to the degree of scheduling flexibility. Among all strategies, PE_W become the best algorithm again with the highest acceptance rate. It presents better performance than Du_B as the flexibility of scheduling increases and defeats other strategies easily throughout the range of values.
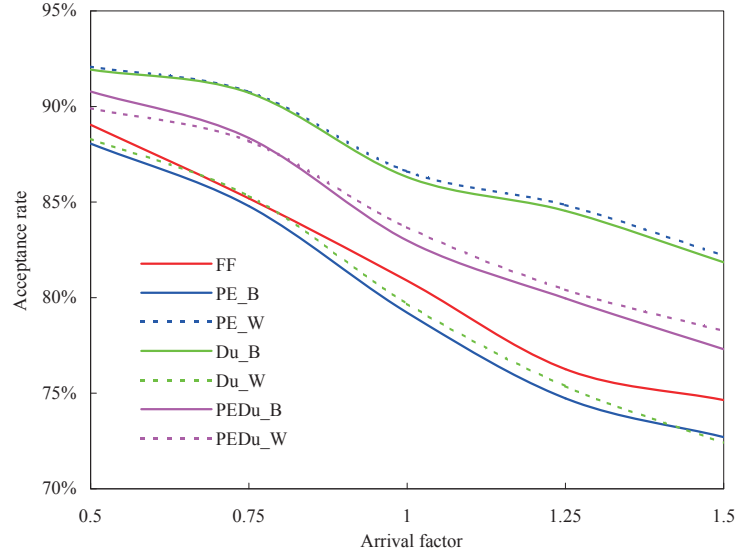
18

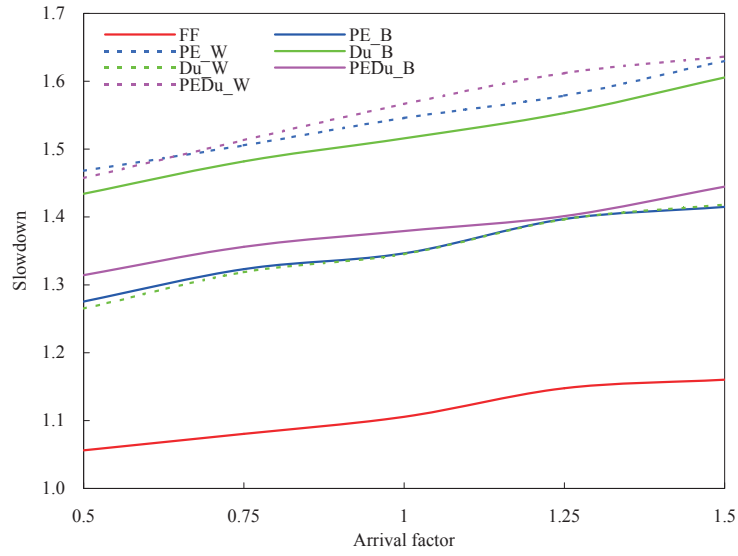Figure 4: Acceptance rate vs arrival factor with UMed=7

Figure 5: Slowdown vs arrival factor with UMed=7

Figure 7 presents the average slowdowns of strategies with different scheduling flexibilities. As the scheduling flexibility increase, the average slowdowns of all strategies increase, which agrees with the intuition that the more flexibility an AR request has in scheduling, the longer the waiting time and larger slowdown will be. Moreover, the relative performance of the curves are similar to the others observed earlier: FF is always the one with smallest values of slowdown by allocating AR jobs to run as early as possible.
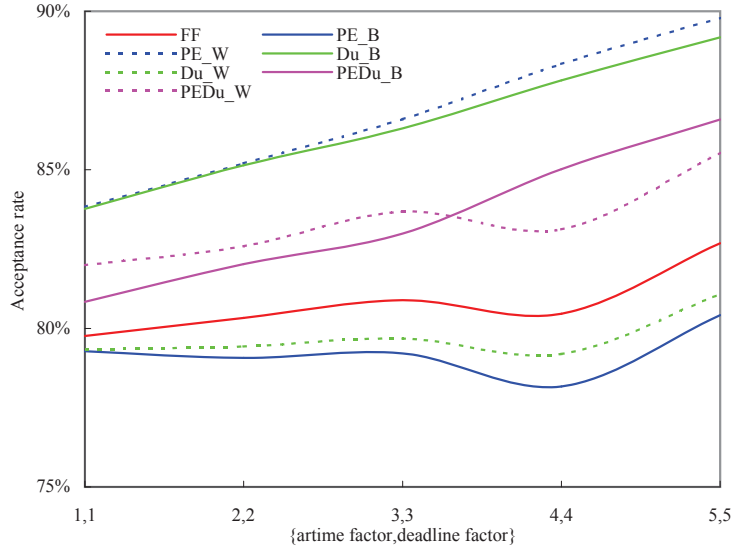

Figure 6: Acceptance rate vs $\{artime factor, deadline factor\}$ with UMed=7

7. Conclusions and discussions

In this paper,we discuss about the scheduling model and algorithms for parallel AR jobs with deadline. We proposed a new data structure and a set of operations to organize the availability of multiprocessor systems for single- and/or multiple-processor advance reservation requests with immediate or general deadline constrains in a way that enables efficient search and update operations, formulated a suite of scheduling policies for the data structure to allocate resources for AR requests, and investigated their performance via simulation. Based on a comprehensive performance evaluation study of the scheduling policies with simulation, it's shown that job sizes
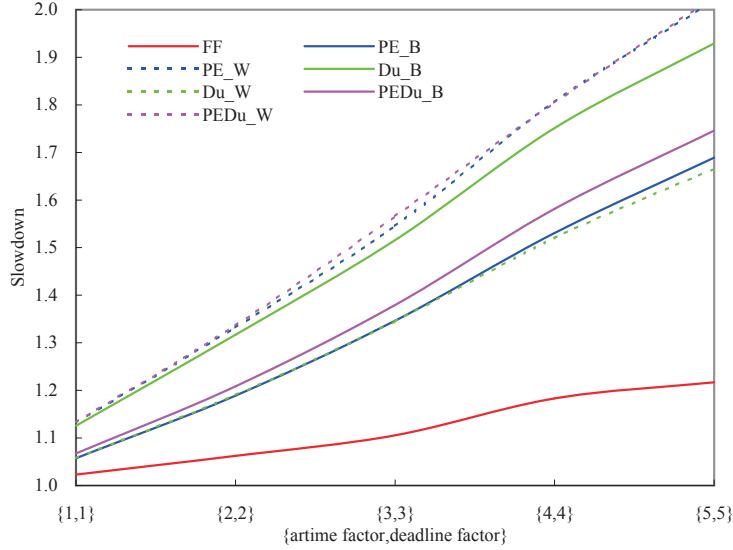
Figure 7: Slowdown vs $\{artime\,factor, deadline\,factor\}$ with UMed=7

and durations, system load and the flexibility of scheduling will impact the performance metrics of all the scheduling algorithms. Among them, the $PE\,Worst\,Fit$ algorithm becomes the best algorithm for the scheduler with the highest acceptance rate of AR requests, and the jobs with the $First\,Fit$ algorithm experience the lowest average slowdown. The simulator and the simulations verified that the data structure, its operations and the scheduling policies are efficient and effective in such computing environments, and can be used in practice. Moreover, because the data structure can support different scheduling policies in a flexible way, other scheduling policies can be easily integrated in the system.

In the research of the data structure,its operations and the scheduling policies, we assume that the resources are homogeneous and the jobs are rigid. However, They can be extended to support heterogeneous resources and malleable jobs in the future. If the system is heterogeneous,i.e., the capacities of the resources in the system are not the same, we can standardize the capacities of the resources and the requirements of the jobs by using a 'standard' resource. In this way, the capacity of each resource and the requirement of each job are described by referring to the standard resource, and we can organize the 'standardized' capacities of the resources in the data

21

structure, and allocate 'standardized resources' for the jobs with 'standardized' requirements. On the other hand, for malleable jobs, their requirements on the number of PEs and durations are not fixed. If a malleable job's requirement of the number of PEs changes, its duration will change along. However, in the $findAllocation(t_r, t_{du}, t_{dl}, n_{job}, policy)$ algorithm in SubSection 4.2, the number of PEs(i.e., n_{job}) and the time-related constraints(i.e., $t_r, t_d u, t_d l$) are rigid. To support malleable jobs, the malleable requirements on the number of PEs and time-related parameters of a job should be 'translated' into a group of rigid ones, then those rigid parameters can be used to find resources for the jobs by using the $findAllocation(t_r, t_{du}, t_{dl}, n_{job}, policy)$ algorithm. Additionally, some new criteria should be designed to choose an allocation for the malleable job among the group of rigid parameters. How to 'translate' the requirements of a malleable AR job with deadline constraint into a group of rigid parameters is also a problem to be considered. In the future, we plan to investigate the problems for heterogeneous resources and malleable jobs in more detail.

Acknowledgements

References

[1] I. Foster, M. Fidler, A. Roy, V. Sander, L. Winkler, End-to-end quality of service for high-end applications, Computer Communications 27 (14) (2004) 1375–1388.

[2] R. Buyya, D. Abramson, J. Giddy, Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid, in: Proceedings of the 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, 2000, pp. 283–290.

[3] R. Al-Ali, O. Rana, D. Walker, S. Jha, S. Sohail, G-qosm: Grid service discovery using qos properties, Computing and Informatics 21 (4) (2002) 363–382.

[4] D. Jackson, Q. Snell, M. Clement, Core algorithms of the maui scheduler, in: Job Scheduling Strategies for Parallel Processing, Springer, 2001, pp. 87–102.

[5] Platform computing corporation.
URL http://www.platform.com

[6] B. Bode, D. Halstead, R. Kendall, Z. Lei, D. Jackson, The portable batch scheduler and the maui scheduler on linux clusters, in: Proceedings of the 4th Annual Linux Showcase and Conference, 2000, pp. 1–9.

[7] C. Castillo, G. Rouskas, K. Harfoush, Online algorithms for advance resource reservations, Journal of Parallel and Distributed Computing DOI: 10.1016/j.jpdc.2011.01.003.

[8] D. Nurmi, R. Wolski, J. Brevik, Probabilistic Reservation Services for Large-Scale Batch-Scheduled Systems, IEEE Systems Journal 3 (1) (2009) 6–24.

[9] K. Aida, H. Casanova, Scheduling mixed-parallel applications with advance reservations, Cluster Computing 12 (2) (2009) 205–220.

[10] C. Castillo, G. Rouskas, K. Harfoush, Resource co-allocation for large-scale distributed environments, in: Proceedings of the 18th ACM international symposium on High performance distributed computing, 2009, pp. 131–140.

[11] Y. Lee, A. Zomaya, Rescheduling for reliable job completion with the support of clouds, Future Generation Computer Systems 26 (8) (2010) 1192–1199.

[12] T. Cucinotta, K. Konstanteli, T. Varvarigou, Advance reservations for distributed real-time workflows with probabilistic service guarantees, in: IEEE International Conference on Service-Oriented Computing and Applications, 2009, pp. 1–8.

[13] B. Li, D. Zhao, Performance impact of advance reservations from the grid on backfill algorithms, in: Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on, IEEE, 2007, pp. 456–461.

[14] Q. Snell, M. Clement, D. Jackson, C. Gregory, The performance impact of advance reservation meta-scheduling, in: Job Scheduling Strategies for Parallel Processing, Springer, 2000, pp. 137–153.

[15] S. Naiksatam, S. Figueira, Elastic reservations for efficient bandwidth utilization in LambdaGrids, Future Generation Computer Systems 23 (1) (2007) 1–22.

[16] M. Margo, K. Yoshimoto, P. Kovatch, P. Andrews, Impact of reservations on production job scheduling, in: Job Scheduling Strategies for Parallel Processing, 2008, pp. 116–131.

[17] L. Burchard, Analysis of data structures for admission control of advance reservation requests, IEEE Transactions on Knowledge and Data Engineering 17 (3) (2005) 413–424.

[18] Q. Xiong, C. Wu, J. Xing, L. Wu, H. Zhang, A linked-list data structure for advance reservation admission control, Networking and Mobile Computing (2005) 901–910.

[19] T. Wang, J. Chen, Bandwidth tree-a data structure for routing in networks with advanced reservations, in: 21st IEEE International Performance, Computing, and Communications Conference, 2002, pp. 37–44.

[20] W. Nie, M. Panahi, K. Lin, A Flexible Schedule Reservation Scheme for Real-Time Service-Oriented Architecture, in: 12th IEEE International Conference on Commerce and Enterprise Computing, IEEE, 2010, pp. 1–8.

[21] R. Brown, Calendar queues: A fast o (1) priority queue implementation for the simulation event set problem, Communications of the ACM 31 (10) (1988) 1220–1227.

[22] A. Sulistio, U. Cibej, S. Prasad, R. Buyya, Garq: An efficient scheduling data structure for advance reservations of grid resources, International Journal of Parallel, Emergent and Distributed Systems 24 (1) (2009) 1–19.

[23] M. Netto, K. Bubendorfer, R. Buyya, Sla-based advance reservations with flexible and adaptive time qos parameters, Service-Oriented Computing–ICSOC 2007 (2010) 119–131.

[24] P. Balakrishnan, T. Somasundaram, SLA enabled CARE resource broker, Future Generation Computer Systems 27 (3) (2010) 265–279.

[25] J. Xu, C. Qiao, J. Li, G. Xu, Efficient burst scheduling algorithms in optical burst-switched networks using geometric techniques, IEEE Journal on Selected Areas in Communications 22 (9) (2004) 1796–1811.

[26] C. Castillo, G. Rouskas, K. Harfoush, On the design of online scheduling algorithms for advance reservations and qos in grids, in: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, 2007, pp. 1–10.

[27] M. Pinedo, Scheduling: theory, algorithms, and systems, Springer Verlag, 2008.

[28] L. Kunrath, C. Westphall, F. Koch, Towards advance reservation in large-scale grids, in: Third International Conference on Systems, 2008, pp. 247–252.

[29] L. Bo, Z. Dongfeng, S. Bin, Simulating platform for grid computing with reservations, Journal of System Simulation 18 (z2) (2006) 373–376.

[30] D. Feitelson, Parallel workloads archive.
URL http://www.cs.huji.ac.il/labs/parallel/workload.

[31] U. Lublin, D. Feitelson, The workload on parallel supercomputers: modeling the characteristics of rigid jobs, Journal of Parallel and Distributed Computing 63 (11) (2003) 1105–1122.

[32] SimJava.
URL http://www.dcs.ed.ac.uk/home/hase/simjava/

[33] F. Heine, M. Hovestadt,O. Kao,A. Streit,On the impact of reservations from the grid on planning-based resource management,Computational Science–ICCS 2005,155–162.